Androbenchmark: high performance computation on android devices

Fabio Gritti
fabio1.gritti@mail.polimi.it

Sebastiano Mariani
sebastiano.mariani@mail.polimi.it

Tutor: Matteo Ferroni
Professors: Donatella Sciuto, Marco D. Santambrogio

Contents

1 Introduction

Nowadays expectations about app's performance on mobile devices are increasing, users want to open their applications and get results in as few time as possible. But how fast Android's apps can go considering that they are written in Java and executed inside a Virtual Machine? We know that Java has the advance of portability,and this is important since Android is the most widespread mobile OS with a huge fragmentation[10] across different devices. But what about the performance? Since Java is knowly not so fast, how can developers speed up applications?
The only possibility before Android 3.0 was to exploit the NDK[2] that allows developers to write code in C or C++ and to interface with their Android applications through the Java Native Interface (JNI) mechanism, but all this at the price of low portability: in fact using jni limits the program's portability at the platforms for which exists an implementation of native library.
Android version 3.0 added a third option:**RenderScript**, which is an API for intensive computation using heterogeneous computing. RenderScript is more widely available across Android devices than the NDK.For example, RenderScript may work on Google TV devices, but there is *currently* no NDK support available for the Google TV platform, even though it runs Android (we will provide a deeper comparison about the two technologies later).
The goal of this project is to analyze the performance gain using this "new" Google's toolchain and to compare it with the Java and JNI solutions; In order to accomplish this we created an application called AndroBenchmark that we will use to launch tests and to collect results, and a server that will receive results from different devices for further global analysis.
We will present first a short description of the technology, then the structure of our app, and finally the results obtained and our final conclusions about the work.

2 Renderscript

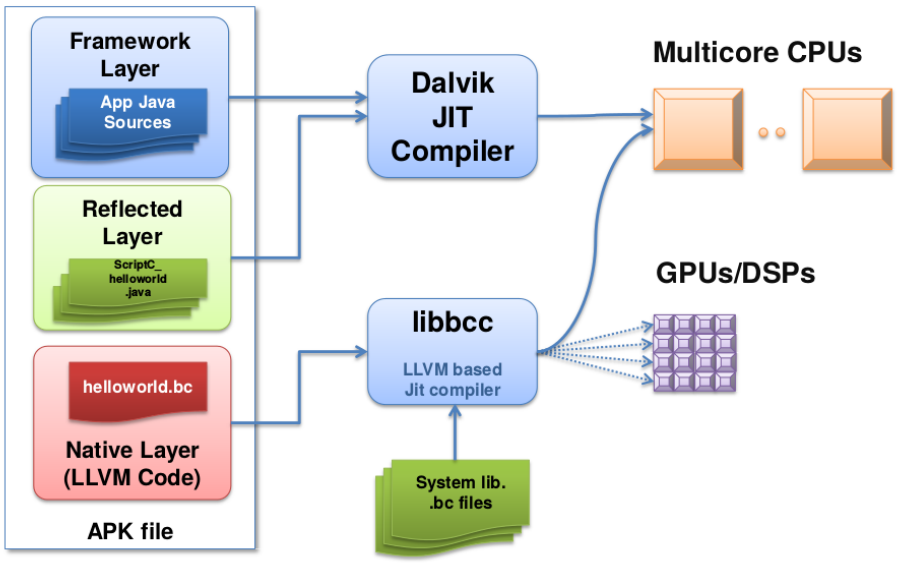
2.1 Overview

RenderScript is a framework (or better, an auxiliary toolchain that consists of Clang compiler and LLVM[3]) for running computationally intensive tasks at high performance on Android; it is primarily oriented for use with data-parallel computation, although (as we will see) serial computationally intensive workloads can benefit as well. When using RenderScript in *parallel mode* the runtime will parallelize a workload across all processors available on a device, such as multi-core CPUs, GPUs, or DSPs, allowing the developer to express algorithms rather than scheduling jobs handling or load balancing. On the other hand using RenderScript in *serial mode* will result in a task scheduled only on one CPU, but with all the benefit that we are going to explain in this paper respect to JNI.
RenderScript is especially useful for applications performing image processing, computational photography, or computer vision, but you can also use it for general purpose tasks.

2.2 How it works

To begin with RenderScript there are two main concepts to keep in mind: the kernels are written in a C99-derived language, while Java API are used to manage the lifetime of RenderScript resources and to control kernel execution; these Java wrappers are automatically generated by the IDE, and are called the **reflected layer**.
RenderScript code is processed by Clang. (a C, C++, Objective C and Objective C++ front-end for the LLVM compiler)and is turned into LLVM bytecode. LLVM is not a virtual machine in a strong sense as it does not "execute" this bytecode.Rather, it provides a second compilation step, turning the LLVM bytecode into machine code for the target resources.
The second compilation step may be performed either on the developer's machine or directly on the device. The first option means that the Clang-LLVM toolchain generates code for 3 processor families: ARM, MIPS and Intel. In this way, the APK file already contains machine code generated from RenderScript code for these processors and no further compilation is needed on the device.
In the second scenario, RenderScript code may run on graphical co-processors and other DSPs. In this case, the on-device LLVM compiler accesses the LLVM bytecode (packaged into the APK file) and will generate machine code for these specialized processors at install time.[3]

The next picture gives you an overview of the toolchain just described.



2.3 Limitations

During our experience with RenderScript and after some researches about it we discovered some limitations too:

1. It is not possible to know if the thread is running on the CPU or GPU
2. There isn't a real possibility of thread synchronization, the only chance is using the *script group*, (a mechanis that allow the developer to connects kernel together in serial and makes the output of a scripts the input of the next one) and consequentelly no possibility to share in a easy manner data across threads.
3. The technology is still acerb and lacks of a good documentation

2.4 Comparison with NDK and OpenCL

For an average Android programmer, RenderScript provides a much more productive way to offload computation-intensive code fragments to highly optimized native code than NDK. RenderScript is integrated within the Android SDK, compilation is super-fast, wrappers are generated automatically, coding parallel execution is simpler than either with the SDK or with the

NDK. Code written with the NDK must be compiled for each target native platform ahead of time. If the app is launched on a platform with an unsupported architecture, the NDK portions of the application will not function properly. RenderScript does a first pass compile on your development machine and then a final pass on the target device, resulting in more efficient native binary code. This means that any device that supports RenderScript will run your code, regardless of the architecture. For pure computational uses, RenderScript setup and configuration is easy and the results may in fact outperform similar implementations that use the NDK, with less coding necessary. RenderScript code may improve in the future as the final compile step improves and more hardware support and compute support are added for GPUs. Native code through the NDK, on the other hand, is unlikely to see performance improvements other than those that come about through hardware changes.Finally applications that leverage the NDK can be line-level debugged using gdb. RenderScript, instead, can't be debugged while it's running. Given the nature of RenderScript and how it works with multiple cores, this isn't a huge surprise, but this can make finding and eliminating bugs more difficult.

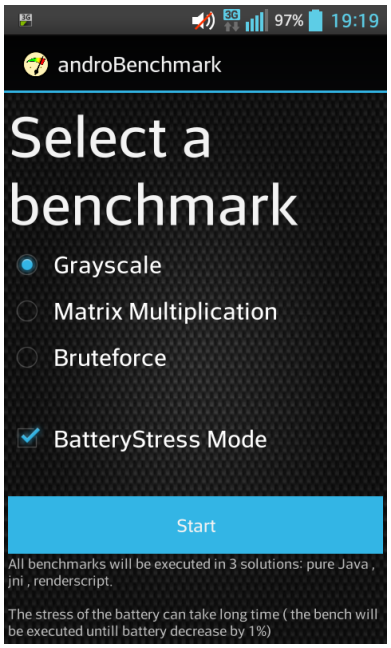
Comparison with OpenCl(another framework for writing programs that execute across heterogeneous platforms: CPUs GPUs,DSPs,FPGAs and other processors) is a big point of discussion on the internet.
First of all, OpenCL uses the execution model first introduced in CUDA,where a kernel is made up of one or many groups of workers, and each group has fast shared memory and synchronization primitives within that group. As a consequence the description of an algorithm has to be intermingled with how that algorithm should be scheduled on a particular architecture.
That's great when you're writing code for one architecture and you want absolute peak performance, but this comes at the expense of performance portability. On mobile, Android needs performance portability between many different GPUs and CPUs vendors with very different architectures[10]. If Android had to rely on a CUDA-style execution model, it would be almost impossible to write a single kernel and have it run acceptably on a range of different devices.
RenderScript abstracts control over scheduling away from the developer at the cost of some peak performance; However there are some point in favor to OpenCl:

- GPU vendors are the driving force behind OpenCL and their tools provide stable, high-speed kernel execution. When GPU technology

improves, OpenCL improves immediately.

- If Android becomes a desktop OS, it should be able to access desktop GPUs and mobile GPUs. OpenCL has broad support on both platforms.
- In addition to CPUs and GPUs, OpenCL kernels can be executed on DSPs and FPGAs. Future high-performance devices will be more likely to support OpenCL than any other language.
- When OpenCL devices are added to a context, they can work together to execute kernels. With OpenCL, embedded devices have the potential of accessing more powerful systems to crunch data.
- If iOS supports OpenCL and Android doesn't, GPU-accelerated apps will run faster on iOS. High-performance mobile computing isn't a big deal yet, but there's no telling what the future may bring.

3 Androbenchmark



3.1 Goals

3.1.1 Comparison of performance

The goal of the app is to collect first of all the execution's time of our three benchmarks executed with the 3 solutions: Java , Jni , RenderScript in order to understand how much a solution performs with respect to the other.
We will use RenderScript in parallel mode for image processing (grayscaling) and serial mode for general purpose tasks (matrix mult. and bruteforcing) to demonstrate that the gain is present in every type of task and mode.
For every benchmarks are performed three tests, the first with a small data size, the second with medium data size and the last with big data size; this is for understand how a solution worsens its execution time due to data size.
AndroBenchmark sends the local results to our server in order to understand if them are always true on all the tested devices.

3.1.2 Testing of resistance

The meaning of the "resistence" is interpreted as: "*if that task is an emergency task in which more loops mean more success, with a given budget of 1% of battery how many loops can that solutions provide to us?*";This isn't a perfect representation of the *power consumption* due to background process, different times of executions and different battery status, but it can give a raw idea of that and provide us interesting results. Before start this type of test we implemented a "junk" function in order to lose 1% of battery from the current level and then start the test in a clean status. We used the data of medium size to perform the benchmark's loops; at the end of every loop we control if the battery's level has lost 1%, if not another loop of the task is performed, if yes we return the total number of loops made since now.
AndroBenchmark sends the local results to our server in order to understand if them are always true on all the tested devices.

3.2 About the benchmarks

In this subsection we are going to describe briefly the tasks that we used as benchmarks in our application.

3.2.1 Grayscale

The Grayscale is the simplest example of image processing. For this benchmark we choose an image made up with a lot of colors in the following sizes:

- 307x230
- 667x500
- 1334x1000

The algorithm in renderscript exploits the parallel mode, that means that jobs will be parallelized on the avalilable CPUs,GPUs,DSPs.
The solutions in Java and JNI are implemented with 2 nested loops that scan each pixel of the images.

3.2.2 Matrix multiplication

For next general purpose tasks, we implemented a serial solution in order to test renderscript's performances in this mode too.
For the Java solution we use the library Jama[9] in order to test if a good algorithm can reach the JNI and RS performance. Jama is a basic linear algebra package for Java, we compared its matrix multiplication's algorithm with a basic implementation of that and discovered that there was a small gain in terms of executions time, so we decided to use it.
The timestamp taken represent the time to generate the two matrix plus the time to perform the multiplication. We used the following matrix size:

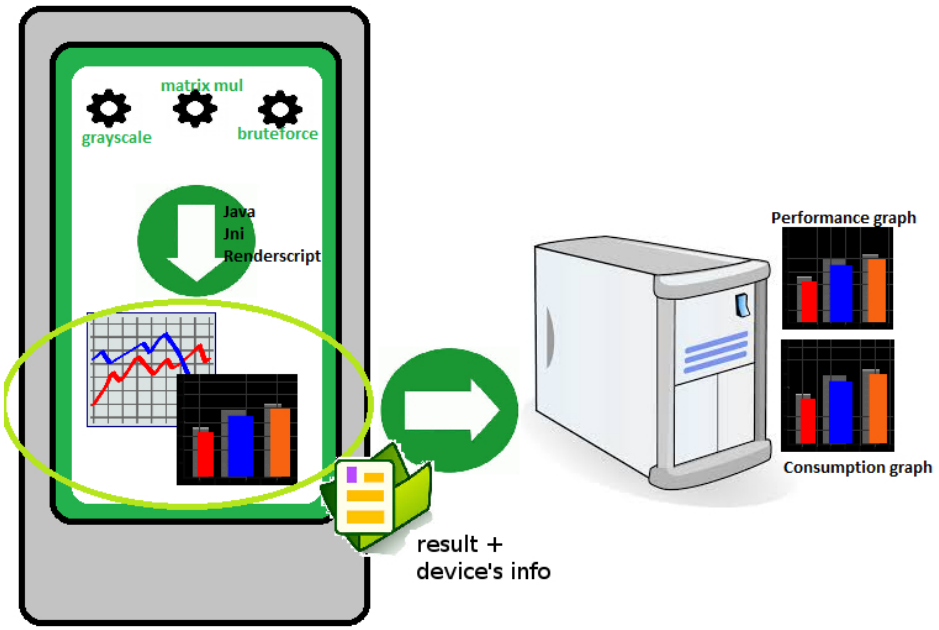
- 100x100
- 150x150
- 200x200

3.2.3 Bruteforce

As title says, we implemented a bruteforce algorithm and tested it with the three technologies.
The length of the words to crack:

- 4 chars
- 5 chars
- 6 chars

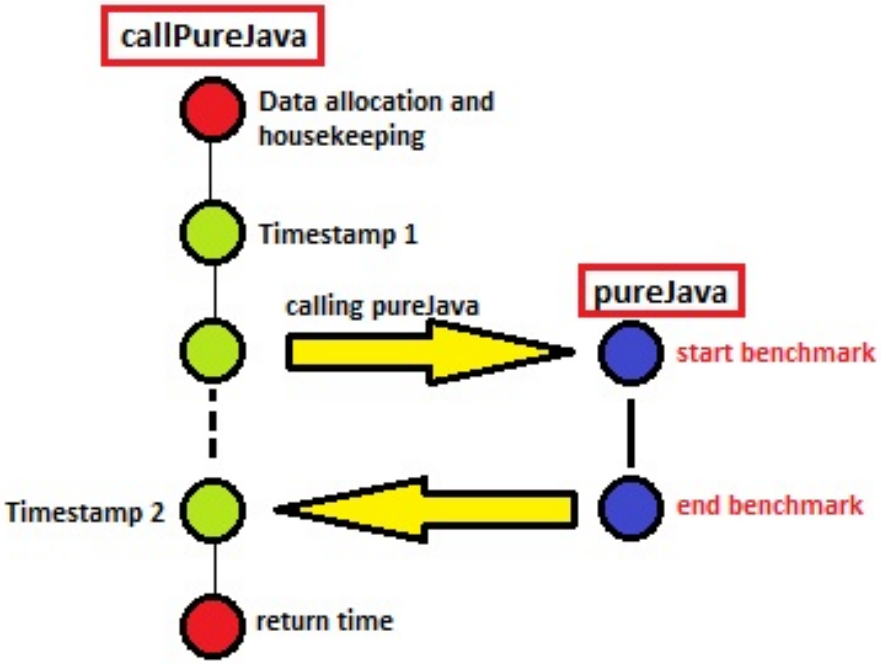
3.3 Application's structure and server



All the benchmarks are executed with an increasing data's size (i.e. in the grayscaling we used three images from the smaller to the bigger) and then the app plots the result in order to compare the execution times. In a second optionally step, it is possible to test the "resistence" of the benchmarks against battery.
Finally, in order to permit us a global analysis, the local results are then sent to our server: this makes a mean of them (per benchmarks) and plots them depending on devices.
This is really useful to understand if the gain is always true and of the same entity on different devices. The server was realized with the NodeJS platform and stores its data in a database implemented with mongoDB.
The following picture is an example of the server's graphs.

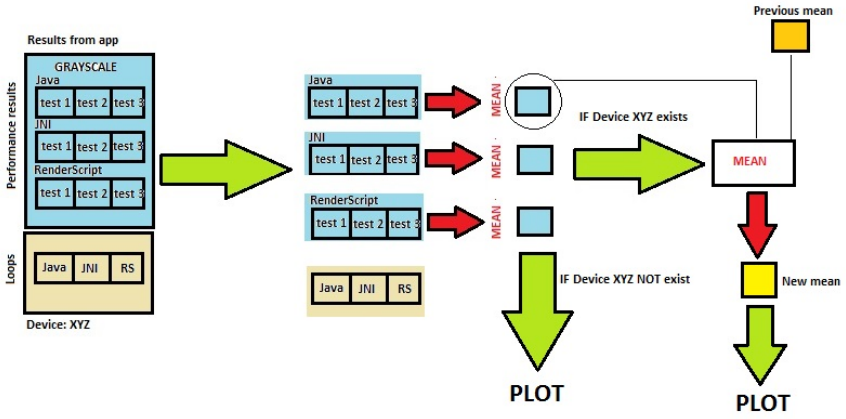
3.3.1 App's code

For every benchmarks we implemented a class that contains the methods to run the algorithm in the three solutions Java, Jni, RenderScript; For every solutions in the class there are two methods: one called, for example, *callPureJava* and another called *PureJava*. The first is the one called by the AsyncTask and the one that allocates structure or any parameter needed. The second is the function called in *CallPureJava* that purely perform the task, the timestamps are taken before the calling of *PureJava* method and when that method returns.
For the plotting of the graphs and the communication with the server we implemented dedicated class. The plotting of graphs is realized with an external library called *androidplot*[4]



3.3.2 Server's code

The server receives data in a JSON file that contains results for each tests of the benchmark (small size, medium size, big size), it makes a mean of all of them and then plots them depending on the device's info contained in the JSON: if the results belong to a new device it creates a new entry in the database and then new bars on the graphs, if not it makes a mean of the new result obtained with the existing values and then replots them.
We decide to plot results of Java, JNI and RenderScript all togheter and for the sake of clarity the graphs with only JNI and RenderScript.



4 Analysys

4.1 Discussion of the result

We discuss here the results obtained per benchmark, first locally on our devices (LGE LG-E460 and Samsung GT-I9300) and then the global trend reported by the server. Finally we will expone our conclusion about the work.
In order to better understand the results, the specification of our two devices are here reported:

LGE LG-E460

- CPU 1 core; ARMv7 Processor rev 10 (v7l)mt6575; Max: 1001,0 MHz; Min: 166,8 MHz
- GPU PowerVR SGX 531; Imagination Technologies; OpenGL ES-CM 1.1; OpenGL ES 2.0 build 1.8@2116011
- RAM 512 MB (483 MB available)
- ANDROID VERSION 4.1.2

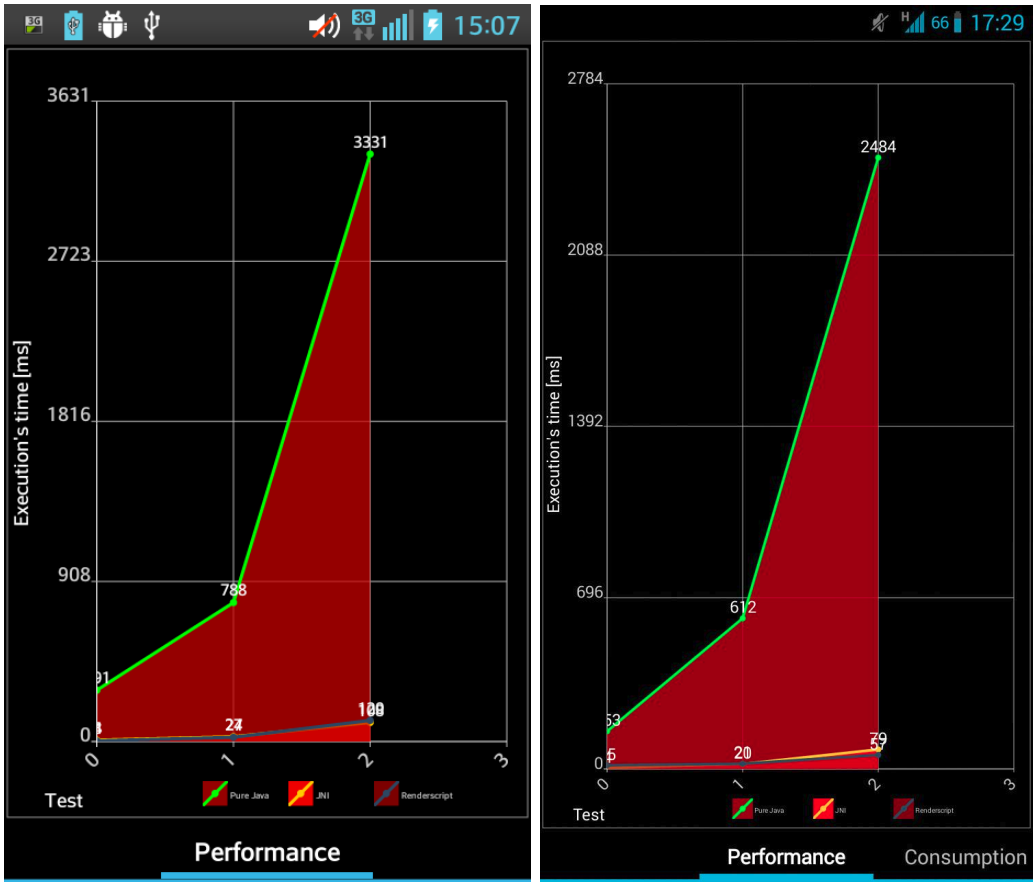
Samsung GT-I9300

- CPU 4 cores; ARMv7 Processor rev 0 (v7l) exynos4; Max: 1800.0 MHz; Min: 200.0 MHz
- GPU Mali-400 MP; ARM; OpenGL ES-CM 1.1; OpenGL ES 2.0

- RAM 1024 MB (1023 MB available)
- ANDROID VERSION 4.2.2

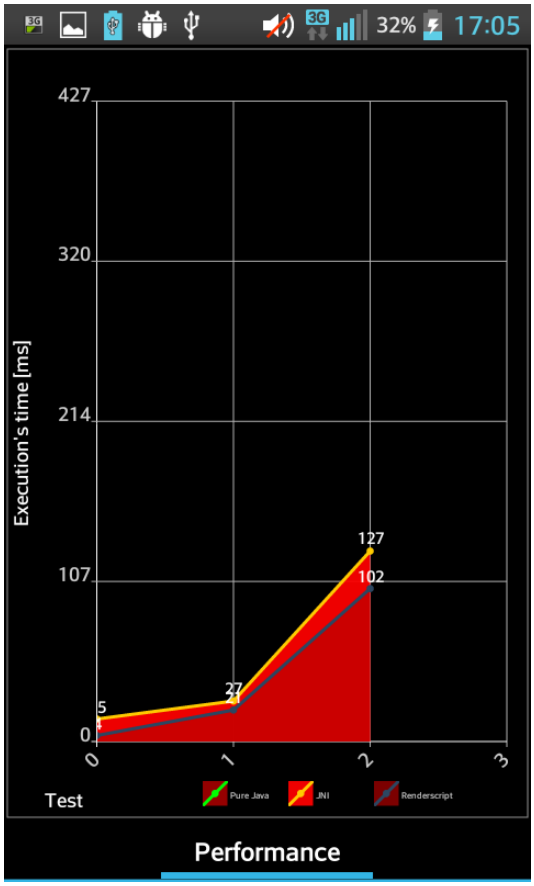
In order to better understand next graphs we are going to provide a small description here: on the X-axis of **performance's graphs** there are the test's number from 0 to 2: 0 for the smallest data size and 2 for the biggest one; on the Y-axis there are the execution's time of the task in milliseconds. The 3 curves represent the results obtained by Java, Jni and RenderScript. About the **consumption's graphs** we find on X-axis the name of the technology, and on the Y-axis the number of loops made with a budget of 1% of battery. The graph on the left side is always referred to the LG platform, while in the right there is the graph of the Samsung device.

4.1.1 Grayscale



About the **local result** the trend is very similar on the two platform (ob-

viously the execution's time is lower in the GT-I9300). Java worsens its performance much faster than the other two solutions while the data becomes larger.
JNI and RenderScript are almost equal in this tests for the data used, but maybe it is better to analyze the situation with a zoomed graph (from the LG device) resulted from the same benchmark but with only RenderScript and JNI plotted:



RenderScript is still better than JNI: as soon as we increase the size of the input data, JNI tends to worsens its performance slightly faster than RenderScript. This behavior is clearly shown in the previous figure, passing from test 1 to test 2.
The following table show us the difference in execution's time passing from test 1 to test 2:

Solutions	dT[ms]
Jni	100
he wins Renderscript	81

Taken as reference the test 2 we can notice the following relations about the executions times of the three technology.
Here are the results from the **LG L5**:

- JNI 26x faster than Java
- RenderScript 33x faster than Java
- RenderScript 1,3x faster than Java

And here the results from the **Galaxy SIII**:

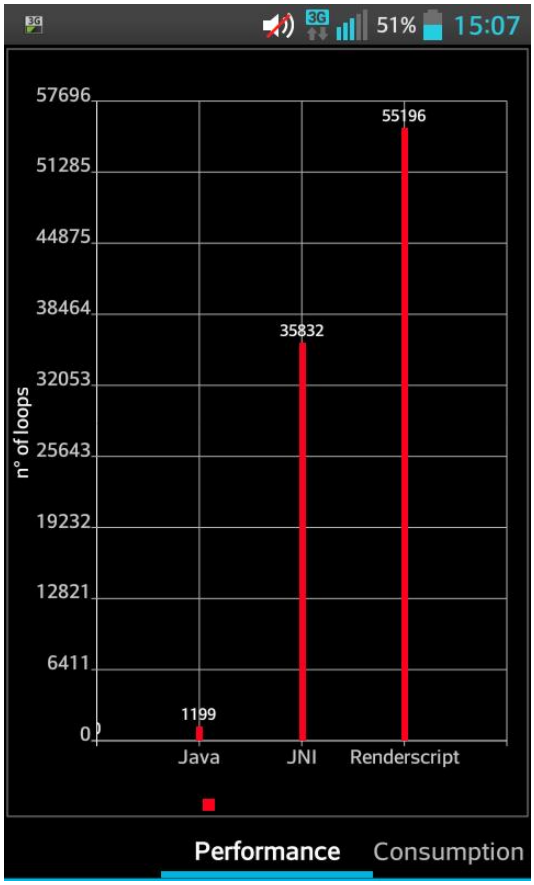
- JNI 31,4x faster than Java
- RenderScript 43,6x faster than Java
- RenderScript 1,4x faster than Java

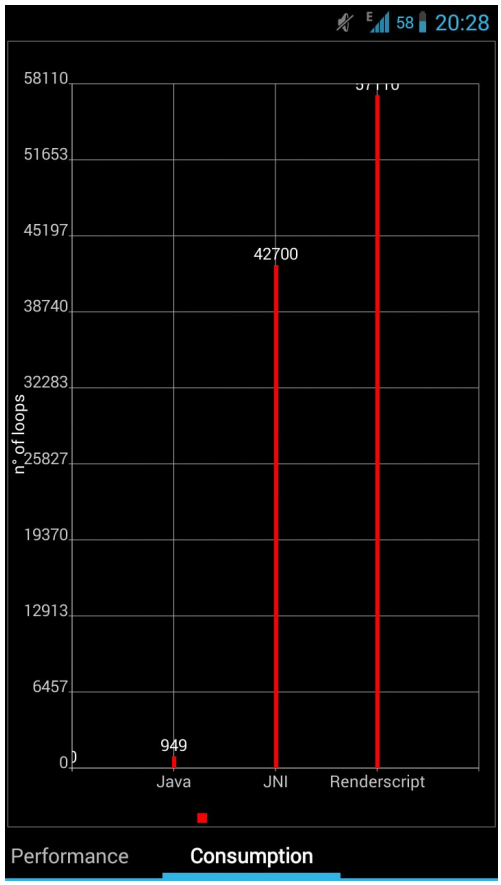
Regarding the number of cycles with 1% of battery's budget, RenderScript wins this match in both the devices as the next graphs show to us.
Numerically the results from the **LG L5** are:

- RenderScript 1,5x JNI loops
- RenderScript 45x Java loops
- JNI 29,9x Java loops

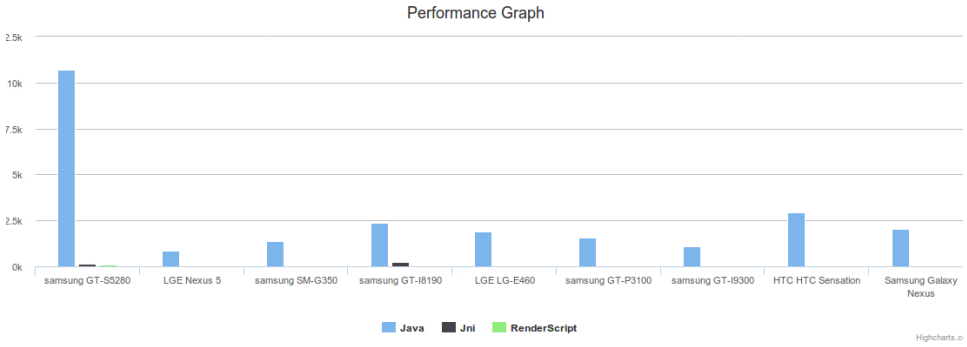
Results from the **Galaxy SIII**:

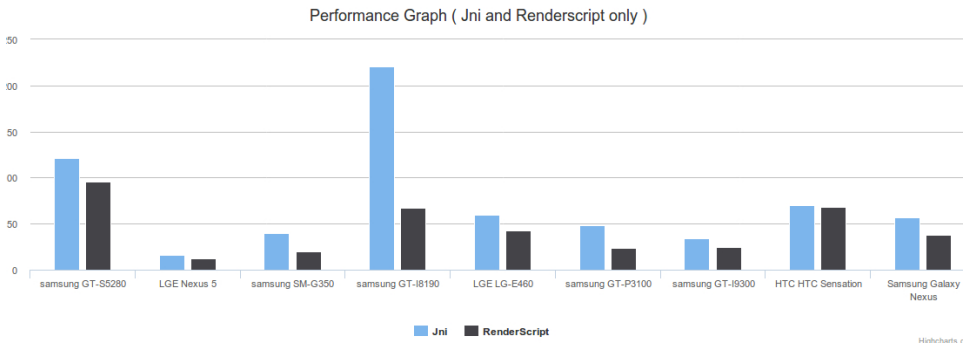
- RenderScript 1,3x JNI loops
- RenderScript 60x Java loops
- JNI 45x Java loops



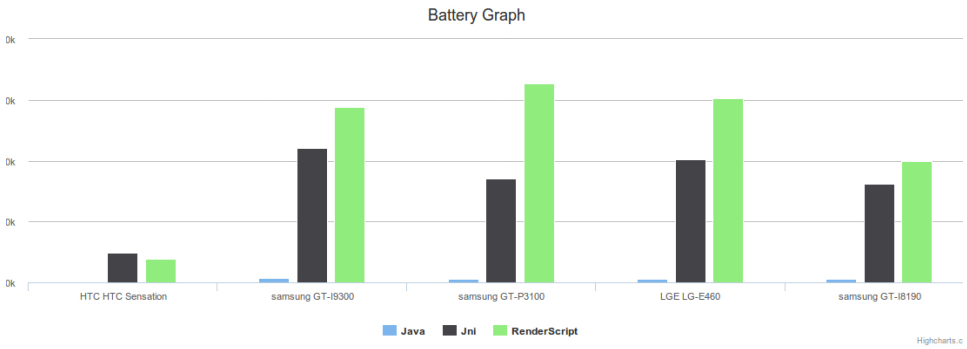


Analyzing the **global trend** reported by the server, we can say that on every platform RenderScript boosts the performances considerably respect to Java and slightly respect to JNI,the local behavoir is then confirmed by the global trend. The following graphs show us first the comparison of all technology (we can't see the bars of JNI and RenderScript beacause their executions times are very low compare to Java's results); and then another graph that compare only JNI and RenderScript on this task.

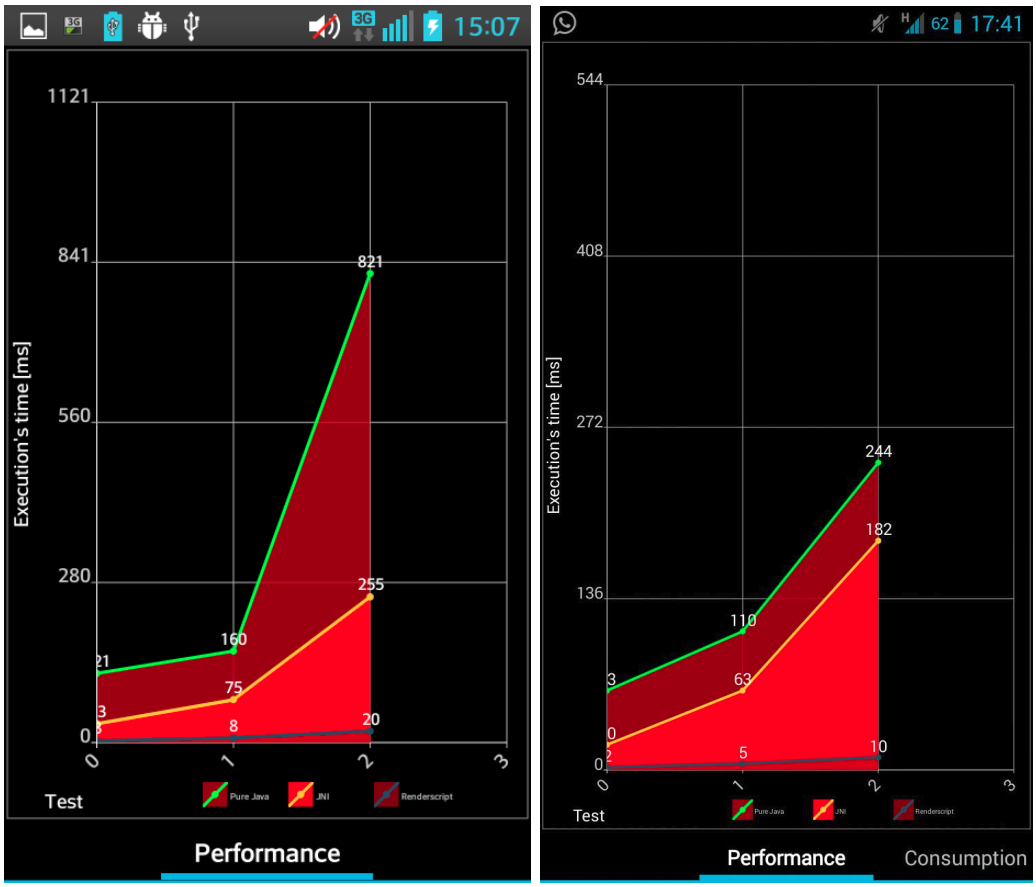




Regarding the cycle, (even if it is influenced a lot by the battery's status and other parameters) RenderScript provide us more loops compared to the other two solutions, with an unique exception for the HTC, where Jni seems wins over renderscript: we think that it is only an exception due to local status of the device.



4.1.2 Matrix multiplication



In this second benchmark we noticed two remarkable things: a serial implementation of an algorithm in RenderScript seems to beat every other solutions, and Java tends to worse its performance faster in the LG platform (left graph).
A deeper analysis points out that RenderScripts tends to worsens its performance visibly lower than the others: the execution times of the solutions while data become bigger increases slower in RenderScript respect to all others. Here we report the difference of times between the second and third tests:

Solutions[LG]	dT[ms]	Solutions[Samsung]	dT[ms]
Java	661	Java	134
Jni	180	Jni	119
Renderscript	12	Renderscript	5

Taken as reference the test 2 we can notice the following relations about the executions times of the three technology.
Here are the results from the **LG L5**:

- JNI 3,2x faster than Java
- RenderScript 41x faster than Java
- RenderScript 12,75x faster than Java

And here the results from the **Galaxy SIII**:

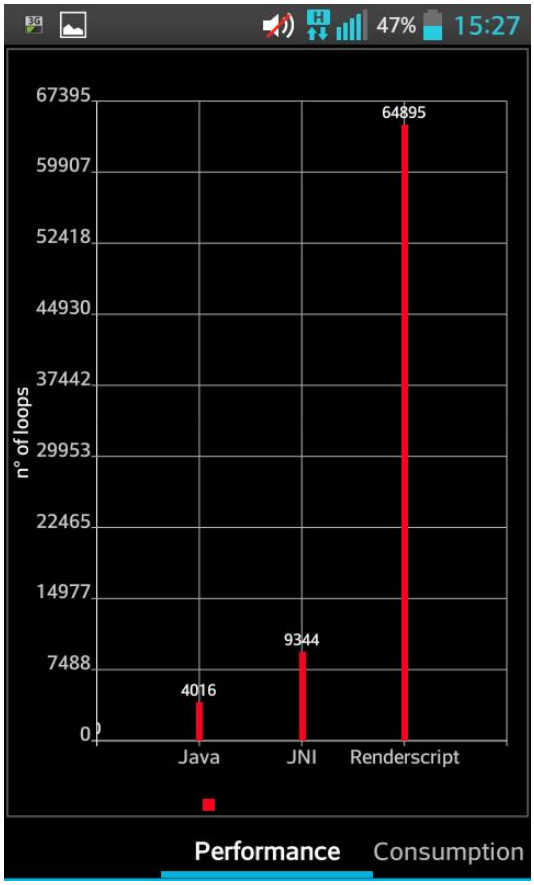
- JNI 1,3x faster than Java
- RenderScript 24,4x faster than Java
- RenderScript 18,2x faster than Java

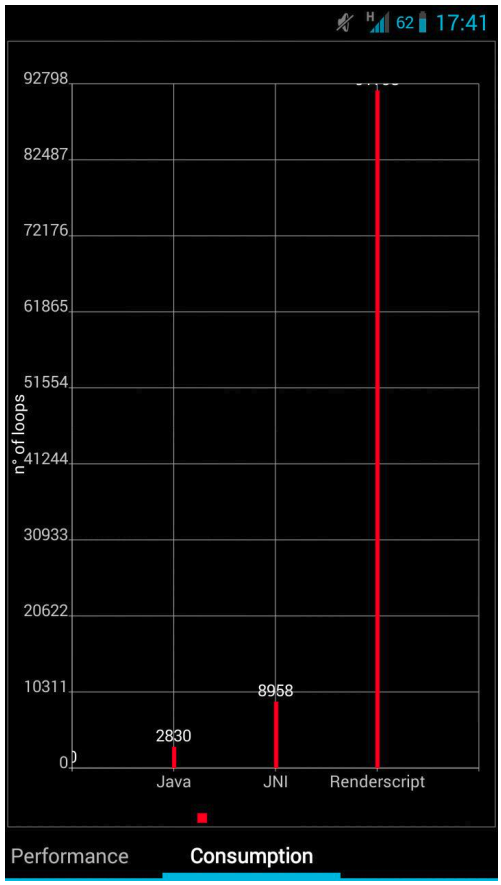
The number of cycles executed rewards RenderScript agains, there is a big difference between RenderScript and JNI, and a huge difference between RenderScript and Java.
Numerically the results from the **LG L5** are:

- RenderScript 6,94x JNI loops
- RenderScript 16,2x Java loops
- JNI 2,3x Java loops

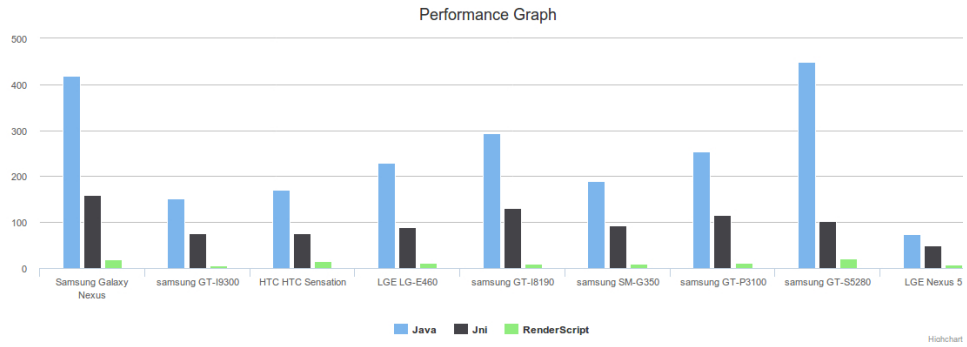
Results from the **Galaxy SIII**:

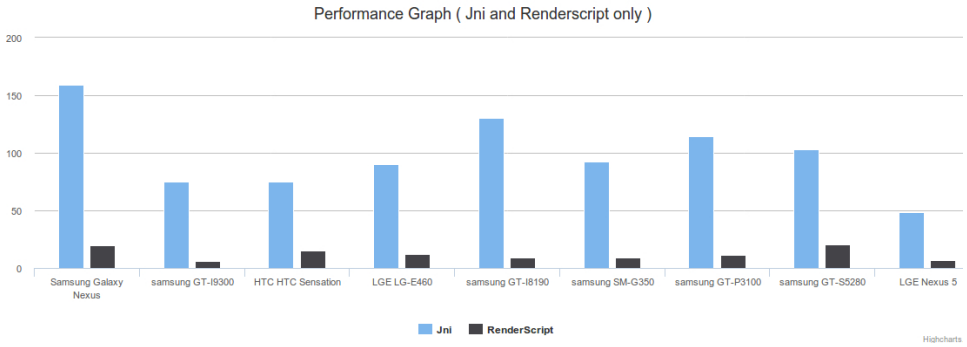
- RenderScript 10,2x JNI loops
- RenderScript 32,4x Java loops
- JNI 3,2x Java loops

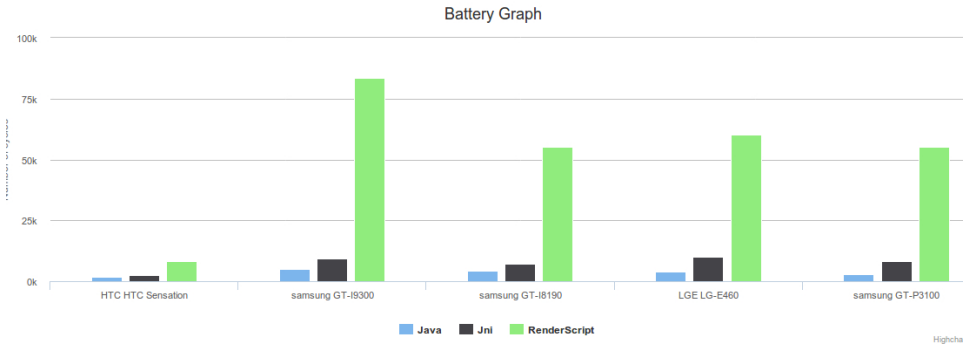




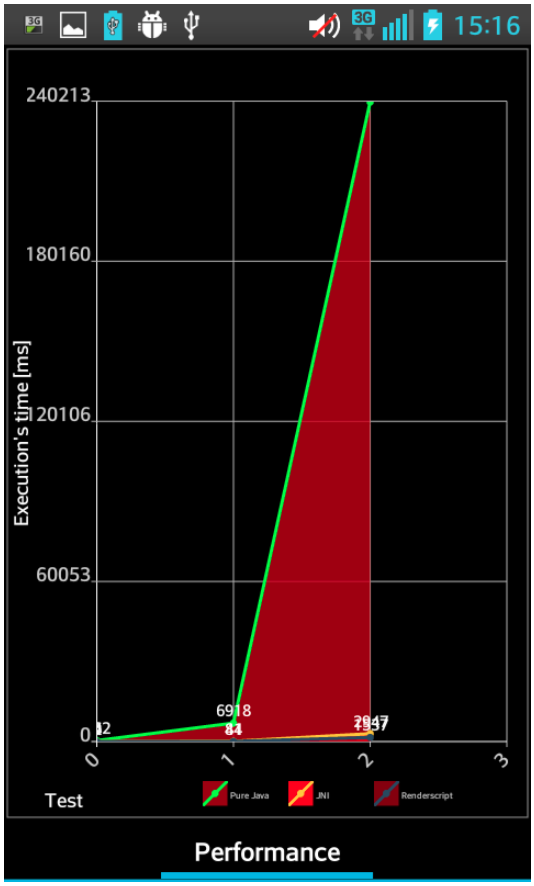
For what concerns **global trend**, according to the following graphs we can say that the local results are confirmed both for the performance and the number of loop executed.It's interesting to notice that the RenderScript's performances remain more or less the same on all the devices, from the oldest to the newest, while java worsens its performance visibly from the newest device to the oldest one: this can suggest various things during the development of an application.


Performance Graph
500
400
300
200
100
0
Samsung Galaxy Nexus
samsung GT-I9300
HTC HTC Sensation
LGE LG-E460
samsung GT-I8190
samsung SM-G350
samsung GT-P3100
samsung GT-S5280
LGE Nexus 5
Java
Jni
RenderScript
Highchart

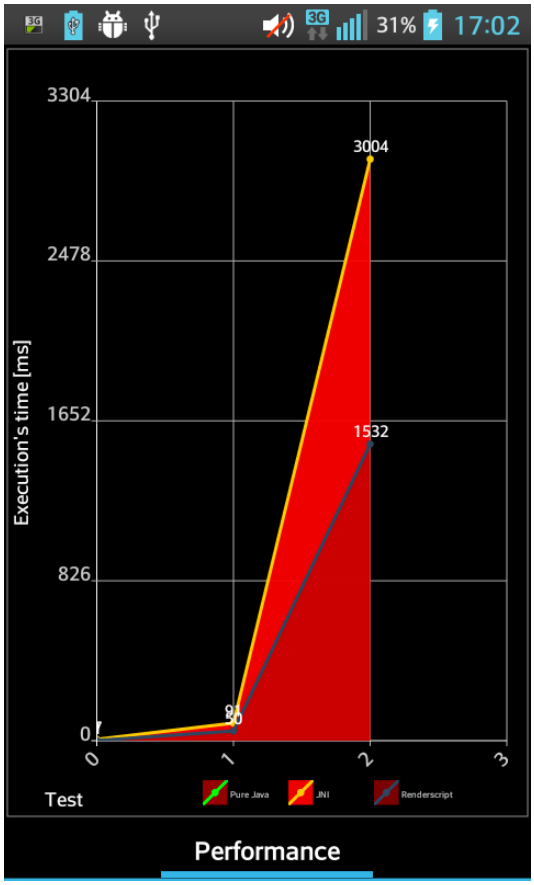

Performance Graph (Jni and Renderscript only)
200
150
100
50
0
Samsung Galaxy Nexus
samsung GT-I9300
HTC HTC Sensation
LGE LG-E460
samsung GT-I8190
samsung SM-G350
samsung GT-P3100
samsung GT-S5280
LGE Nexus 5
Jni
RenderScript
Highcharts


Battery Graph
100k
75k
50k
25k
0k
HTC HTC Sensation
samsung GT-I9300
samsung GT-I8190
LGE LG-E460
samsung GT-P3100
Java
Jni
RenderScript
Highcha

4.1.3 Bruteforce



This last test shows the great power of JNI and RenderScript against the Java solution.As we can see with the big data test(2) the performance of Java are very bad compared to the other two.
The trend is the same on both the devices. For the comparison between JNI and RenderScript is better to analyze a dedicated graph(from LG device):



With this zoom we can notice very clearly that JNI worsens its performance faster than RenderScript; For the sake of clarity, we reported here the usual difference of time in milliseconds from medium to big data:

Solutions	**dT[ms]**
Jni	2913
Renderscript	1482

Taken as reference the test 2 we can notice the following relations about the executions times of the three technology.
Here are the results from the **LG L5**:

- JNI 81,5x faster than Java
- RenderScript 154,3x faster than Java

- RenderScript 1,6x faster than Java

And here the results from the **Galaxy SIII**:

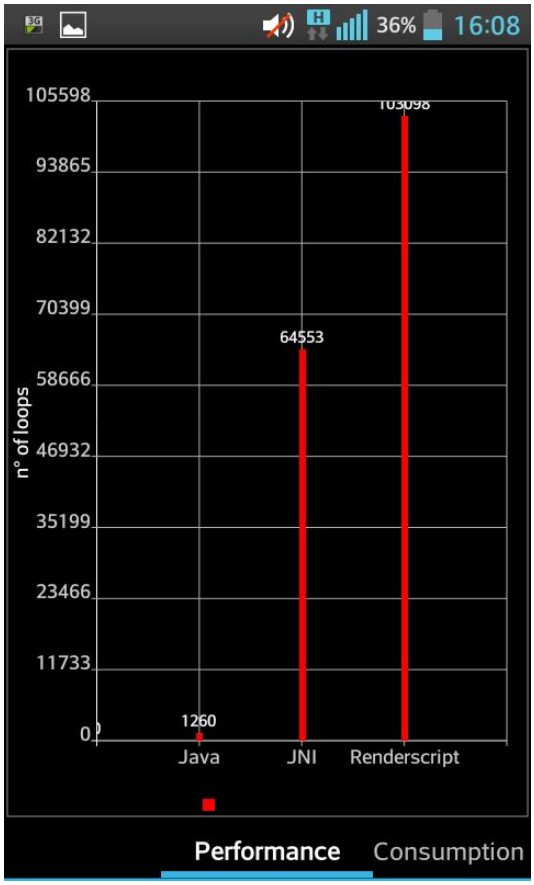
- JNI 25,2x faster than Java
- RenderScript 51,1x faster than Java
- RenderScript 2x faster than Java

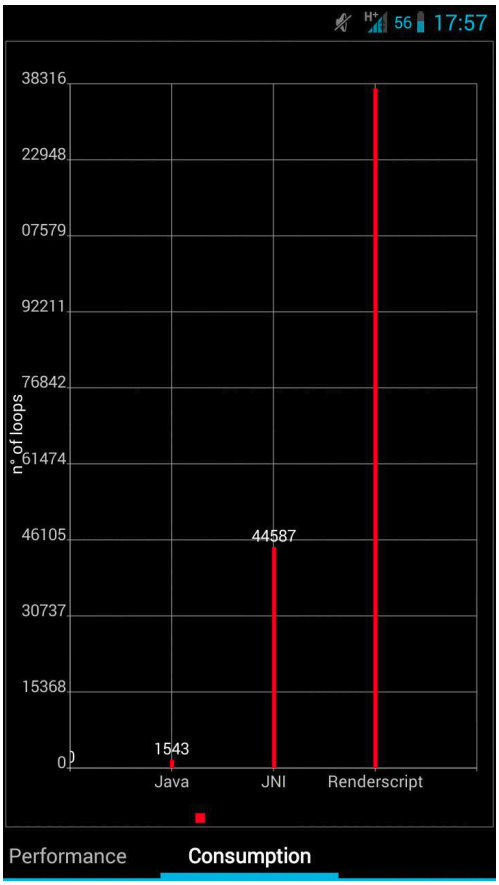
Again, the number of loops performed by RenderScript with the battery's budget given is very large compared to the other two solutions, with Java always at the third position with very few of them.
Numerically the results from the **LG L5** are:

- RenderScript 1,6x JNI loops
- RenderScript 81,8x Java loops
- JNI 51,2x Java loops

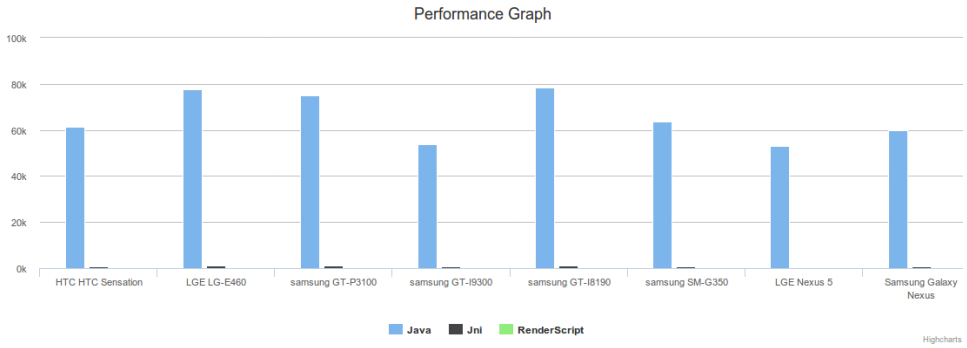
Results from the **Galaxy SIII**:

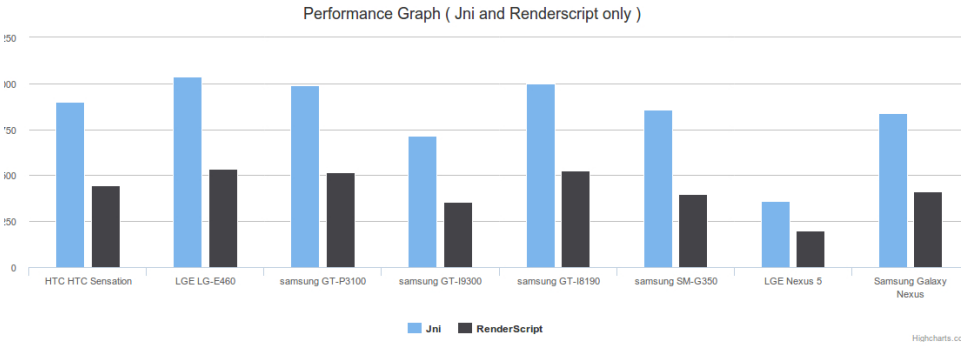
- RenderScript 3x JNI loops
- RenderScript 89,6x Java loops
- JNI 28,9x Java loops

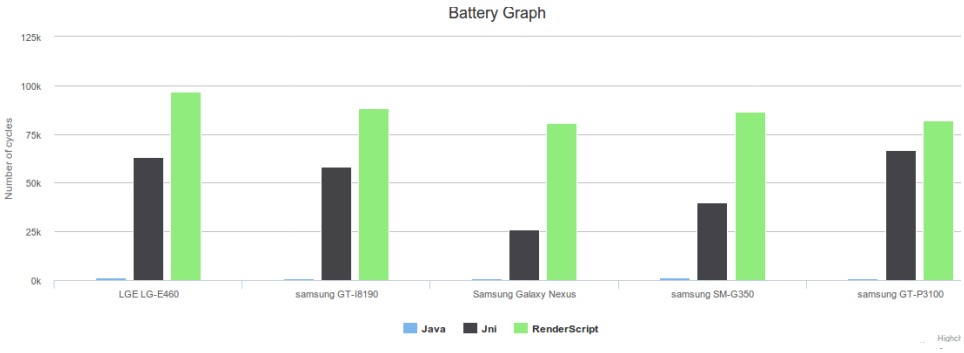




According to next graphs the **global trend** confirms as expected the local results obtained.







5 Conclusion

According to our results, we can conclude that RenderScript is the best solution to optimize the execution of particoular tasks with respect to Java, indipendentely from the device, and it tends to lose its benefit *more slower* than Jni. Furthermore it seems that renderscript is able to mantain its performance on old devices as on newest ones.
With the stress tests performed we demostrated also that it can provide more loops with a restrict budget of battery with respect to the others. As said first, the usage of RenderScript is more supported and easy respect to JNI: the possibility to parallelize code is nearly free (despite some work has still to be done to make it more flexible) and it can be applied to serial tasks too, with always a great benefit.
In a future of more heterogeneous and complex devices, RenderScript will boost the performances of mobile apps at no cost.

References

[1] developer.android.com

[2] developer.android.com/tools/sdk/ndk/index.html

[3] mylifewithandroid.blogspot.it

[4] androidplot.com/

[5] llvm.org

[6] clang.llvm.org/

[7] khronos.org/opencl/

[8] developer.com

[9] math.nist.gov/javanumerics/jama/

[10] opensignal.com/reports/fragmentation.php

[11] [BOOK]Pro Android Apps Performance Optimization

[12] [REPO]bitbucket.org/necst/gritti-mariani-hpps2014-androbenchmark